Code Optimization

- There is no fully optimizing compiler $O$
- Let’s assume $O$ exists: it takes a program P and produces output $\text{Opt}(P)$ which is the smallest possible
- Imagine a program Q that produces no output and never terminates, then $\text{Opt}(Q)$ could be:
  \[
  \text{L1: goto L1}
  \]
- Then to check if a program P never terminates on some inputs, check if $\text{Opt}(P(i))$ is equal to $\text{Opt}(Q)$
- Full Employment Theorem for Compiler Writers, see Rice(1953)
Optimizations

• Non-Optimizations
• Correctness of optimizations
  – Optimizations must not change the meaning of the program
• Types of optimizations
  – Local optimizations
  – Global dataflow analysis for optimization
  – Static Single Assignment (SSA) Form
• Amdahl’s Law

Non-Optimizations

enum { GOOD, BAD }; extern int test_condition();
void check() {
  int rc;
  rc = test_condition();
  if (rc != GOOD) {
    exit(rc);
  }
}

Which version of check runs faster?
Types of Optimizations

• High-level optimizations
  – function inlining
• Machine-dependent optimizations
  – e.g., peephole optimizations, instruction scheduling
• Local optimizations or Transformations
  – within basic block

Types of Optimizations

• Global optimizations or Data flow Analysis
  – across basic blocks
  – within one procedure (intraprocedural)
  – whole program (interprocedural)
  – pointers (alias analysis)
Maintaining Correctness

- What does this program output?

  3

  Not:

  $ decafcc byzero.decaf

  Floating exception

void main() {
    int x;
    if (false) {
        x = 3/(3-3);
    } else {
        x = 3;
    }
    callout(“print_int”, x);
}

branch delay slot (cf. load delay slot)

Peephole Optimization

- Redundant instruction elimination
  - If two instructions perform that same function and are in the same basic block, remove one
  - Redundant loads and stores
    li $t0, 3
    li $t0, 4
  - Remove unreachable code
    li $t0, 3
    goto L2
    ... (all of this code until next label can be removed)
Peephole Optimization

- Flow control optimization
  goto L1
  L1: goto L2
- Algebraic simplification
- Reduction in strength
  - Use faster instructions whenever possible
- Use of Machine Idioms
- Filling delay slots

Constant folding & propagation

- Constant folding
  - compute expressions with known values at compile time
- Constant propagation
  - if constant assigned to variable, replace uses of variable with constant unless variable is reassigned
Constant folding & propagation

• Copy Propagation

\[
\begin{align*}
  a &:= d + e \\
  b &:= d + e \\
  c &:= d + e \\
  t &:= d + e \\
  a &:= t \\
  b &:= t \\
  c &:= t 
\end{align*}
\]

Transformations

• Structure preserving transformations
• Common subexpression elimination

\[
\begin{align*}
  a &:= b + c \\
  b &:= a - d \\
  c &:= b + c \\
  d &:= a - d \ (\Rightarrow b)
\end{align*}
\]
Transformations

- Dead-code elimination (combines copy propagation with removal of unreachable code)

  ```
  if (debug) { f(); } /* debug := false (as a constant) */
  if (false) { f(); } /* constant folding */
  using deadcode elimination, code for f() is removed
  x := t3
  t4 := x becomes t4 := t3
  ```

Transformations

- Renaming temporary variables
  
  \[ t1 := b+c \] can be changed to \[ t2 := b+c \]
  
  replace all instances of \( t1 \) with \( t2 \)

- Interchange of statements

  \[ t1 := b+c \] \[ t2 := x+y \]
  
  \[ t2 := x+y \] can be converted to \[ t1 := b+c \]
Transformations

- Algebraic transformations
  \[ d := a + 0 \implies a \]
  \[ d := d \times 1 \implies \text{eliminate} \]
- Reduction of strength
  \[ d := a \times a \]

Control Flow Graph (CFG)

```c
int main() {
    extern int f(int);
    int i;
    int *a;
    for (i = 0; i < 10; i = i + 1) {
        a[i] = f(i);
    }
}
```
Control Flow Graph in TAC

main:
   i = 0
L0:
   t1 = 10
   t2 = i < t1
   ifFalse t2 Goto L1
   t3 = 4
   t4 = t3 + i
   t5 = a + t4
   param i
t6 = call f, 1
   pop 4
   *(t5) = t6
   t7 = 1
   i = i + t7
   goto L0
L1:
   return

Dataflow Analysis

- \( S \rightarrow id := E \)
- \( S \rightarrow S ; S \)
- \( S \rightarrow \text{if } E \text{ then } S \text{ else } S \)
- \( S \rightarrow \text{do } S \text{ while } E \)
- \( E \rightarrow id + id \)
- \( E \rightarrow id \)
Dataflow Analysis

S ; S  if E then S else S  do S while E

Reaching definitions

d: a := b+c

\[
\text{gen}[S] = \{ d \} \\
\text{kill}[S] = \text{def}(a) - \{ d \} \\
\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])
\]
Reaching definitions

\[ \text{gen}[S] = \text{gen}[S2] \cup (\text{gen}[S1] - \text{kill}[S2]) \]

\[ \text{kill}[S] = \text{kill}[S2] \cup (\text{kill}[S1] - \text{gen}[S2]) \]

\[ \text{in}[S1] = \text{in}[S] \]

\[ \text{in}[S2] = \text{out}[S1] \]

\[ \text{out}[S] = \text{out}[S2] \]

Reaching definitions

\[ \text{gen}[S] = \text{gen}[S1] \cup \text{gen}[S2] \]

\[ \text{kill}[S] = \text{kill}[S1] \cap (\text{kill}[S1] - \text{gen}[S2]) \]

\[ \text{in}[S1] = \text{in}[S] \]

\[ \text{in}[S2] = \text{in}[S] \]

\[ \text{out}[S] = \text{out}[S1] \cup \text{out}[S2] \]
Reaching definitions

\[ \text{gen}[S] = \text{gen}[S1] \]
\[ \text{kill}[S] = \text{kill}[S1] \]
\[ \text{in}[S1] = \text{in}[S] \cup \text{gen}[S1] \]
\[ \text{out}[S] = \text{out}[S1] \]

Iteratively find \text{out}[S] (fixed point)

\[ \text{out}[S1] = \text{gen}[S1] \cup (\text{in}[S1] - \text{kill}[S1]) \]

Reaching definitions

\[ B1 \]
\[ \begin{align*}
  d1: & i := m-1 \\
  d2: & j := n \\
  d3: & a := u1
\end{align*} \]
\[ \text{gen}[B1] = \{ d1, d2, d3 \} \]
\[ \text{kill}[B1] = \{ d4, d5, d6, d7 \} \]

\[ B2 \]
\[ \begin{align*}
  d4: & i := i+1 \\
  d5: & j := j-1
\end{align*} \]
\[ \text{gen}[B2] = \{ d4, d5 \} \]
\[ \text{kill}[B2] = \{ d1, d2, d7 \} \]

\[ B3 \]
\[ d6: a := u2 \]
\[ \text{gen}[B3] = \{ d6 \} \]
\[ \text{kill}[B3] = \{ d3 \} \]

\[ B4 \]
\[ d7: i := u3 \]
\[ \text{gen}[B4] = \{ d7 \} \]
\[ \text{kill}[B4] = \{ d1, d4 \} \]
Reaching definitions

\[ \text{Reaching definitions} \]

\[ \text{B1:} \quad \begin{align*}
  \text{d1: } i & := m-1 \\
  \text{d2: } j & := n \\
  \text{d3: } a & := u1
\end{align*} \]

\[ \text{B2:} \quad \begin{align*}
  \text{d4: } i & := i+1 \\
  \text{d5: } j & := j-1
\end{align*} \]

\[ \text{B3:} \quad \begin{align*}
  \text{d6: } a & := u2
\end{align*} \]

\[ \text{B4:} \quad \begin{align*}
  \text{d7: } i & := u3
\end{align*} \]

\[ \text{gen[B1]} = \{ \text{d1, d2, d3} \} \]
\[ \text{kill[B1]} = \{ \text{d4, d5, d6, d7} \} \]

\[ \text{gen[B2]} = \{ \text{d4, d5} \} \]
\[ \text{kill[B2]} = \{ \text{d1, d2, d7} \} \]

\[ \text{gen[B3]} = \{ \text{d6} \} \]
\[ \text{kill[B3]} = \{ \text{d3} \} \]

\[ \text{gen[B4]} = \{ \text{d7} \} \]
\[ \text{kill[B4]} = \{ \text{d1, d4} \} \]

\[ \text{in[B2]} = \text{out[B1]} \cup \text{out[B3]} \cup \text{out[B4]} \]

\[ \text{out[B2]} = \text{gen[B2]} \cup (\text{in[B3]} - \text{kill[B2]}) \]
\[ \text{out[B2]} = \text{gen[B2]} \cup (\text{in[B4]} - \text{kill[B2]}) \]
Dataflow Analysis

- Compute Dataflow Equations over Control Flow Graph
  - Reaching Definitions (**Forward**)
    \[ \text{out}[BB] := \text{gen}[BB] \cup (\text{in}[BB] – \text{kill}[BB]) \]
    \[ \text{in}[BB] := \cup \text{out}[s] : \forall s \in \text{pred}[BB] \]
  - Liveness Analysis (**Backward**)
    \[ \text{in}[BB] := \text{use}[BB] \cup (\text{out}[BB] – \text{def}[BB]) \]
    \[ \text{out}[BB] := \cup \text{in}[s] : \forall s \in \text{succ}[BB] \]

- Computation by fixed-point analysis

SSA Form

- *def-use* chains keep track of where variables were defined and where they were used
- Consider the case where each variable has only one definition in the intermediate representation
- One static definition, accessed many times
- Static Single Assignment Form (SSA)
SSA Form

• SSA is useful because
  – Dataflow analysis and optimization is simpler when each variable has only one definition
  – If a variable has N uses and M definitions (which use N+M instructions) it takes N*M to represent def-use chains
  – Complexity is the same for SSA but in practice it is usually linear in number of definitions
  – SSA simplifies the register interference graph

SSA Form

• Original Program

  a := x + y
  b := a - 1
  a := y + b
  b := x * 4
  a := a + b

• SSA Form

  a1 := x + y
  b1 := a1 - 1
  a2 := y + b1
  b2 := x * 4
  a3 := a2 + b2

what about conditional branches?
SSA Form

1: \( b := M[x] \)
   \( a := 0 \)

2: if \( b < 4 \)

3: \( a := b \)

4: \( c := a + b \)

1: \( b1 := M[x1] \)
   \( a1 := 0 \)

2: if \( b1 < 4 \)

3: \( a2 := b1 \)

4: \( a3 := \phi(a2, a1) \)
   \( c1 := a3 + b1 \)

SSA Form

1: \( a := 0 \)

2: \( b := a + 1 \)
   \( c := c + b \)
   \( a := b \times 2 \) if \( a < N \)

3: \( \text{return } c \)

1: \( a1 := 0 \)

2: \( a3 := \phi(a2, a1) \)
   \( b1 := \phi(b0, b2) \)
   \( c2 := \phi(c0, c1) \)
   \( b2 := a3 + 1 \)
   \( c1 := c2 + b2 \)
   \( a2 := b2 \times 2 \) if \( a2 < N \)

3: \( \text{return } c1 \)
Optimizations using SSA

• SSA form contains statements, basic blocks and variables
• Dead-code elimination
  – if there is a variable \( v \) with no uses and \( \text{def} \) of \( v \) has no side-effects, delete statement defining \( v \)
  – if \( z := \phi(x, y) \) then eliminate this stmt if no \( \text{defs} \) for \( x, y \)

Optimizations using SSA

• Constant Propagation
  – if \( v := c \) for some constant \( c \) then replace \( v \) with \( c \) for all uses of \( v \)
  – \( v := \phi(c1, c2, ..., cn) \) where all \( c_i \) are equal to \( c \) can be replaced by \( v := c \)
Optimizations using SSA

- Conditional Constant Propagation
  - In previous flow graph, is j always equal to 1?
  - If j = 1 always, then block 6 will never execute and so j := i and j := 1 always
  - If j > 20 then block 6 will execute, and j := k will be executed so that eventually j > 20
  - Which will happen? Using SSA we can find the answer.
Optimizations using SSA

1: $i_1 := 1$  \hspace{1cm} j_1 := 1 \\
\hspace{1cm} k_1 := 0

2: $j_2 := \phi(j_4, j_1)$  \\
$\hspace{1cm} k_2 := \phi(k_4, k_1)$  \\
\hspace{1cm} if $k_2 < 100$

3: if $j_2 < 20$

4: return $j_2$

5: $j_3 := i_1$  \\
$\hspace{1cm} k_3 := k_2 + 1$

6: $j_5 := k_2$  \\
$\hspace{1cm} k_5 := k_2 + 1$

7: $j_4 := \phi(j_3, j_5)$  \\
$\hspace{1cm} k_4 := \phi(k_3, k_5)$

---

After Constant Propagation
Optimizations using SSA

After Constant Propagation

1:

2: \( k_2 := \phi(k_4, 0) \) if \( k_2 < 100 \)

3:

4: return 1

5: \( k_3 := k_2 + 1 \)

7: \( k_4 := \phi(k_3) \)

After Removing Empty Blocks and 1-arg \( \phi \) functions

Optimizations using SSA

1:

2: \( k_2 := \phi(k_3, 0) \) if \( k_2 < 100 \)

5: \( k_3 := k_2 + 1 \)

4: return 1
Optimizations using SSA

• Arrays, Pointers and Memory
  – For more complex programs, we need dependencies: how does statement B depend on statement A?
  – **Read after write**: A defines variable \( v \), then B uses \( v \)
  – **Write after write**: A defines \( v \), then B defines \( v \)
  – **Write after read**: A uses \( v \), then B defines \( v \)
  – **Control**: A controls whether B executes

• Memory dependence
  \[
  M[i] := 4 \\
x := M[j] \\
M[k] := j
  \]
• We cannot tell if \( i, j, k \) are all the same value which makes any optimization difficult
• Similar problems with Control dependence
• SSA does not offer an easy solution to these problems
SSA Form

• Conversion from a Control Flow Graph (created from TAC) into SSA Form is not trivial

• Two famous algorithms:
  – Lengauer-Tarjan algorithm (see the Tiger book by Andrew W. Appel for more details)
  – Harel algorithm

More on Optimization

• *Advanced Compiler Design and Implementation* by Steven S. Muchnick

• Control Flow Analysis
• Data Flow Analysis
• Dependence Analysis
• Alias Analysis
• Early Optimizations
• Redundancy Elimination

• Loop Optimizations
• Procedure Optimizations
• Code Scheduling (pipelining)
• Low-level Optimizations
• Interprocedural Analysis
• Memory Hierarchy
Amdahl’s Law

- \( \text{Speedup}_{\text{total}} = \left( (1 - \text{Time}_{\text{Fraction optimized}}) + \frac{\text{Time}_{\text{Fraction optimized}}}{\text{Speedup}_{\text{optimized}}}-1 \right) \)

- Optimize the common case, 90/10 rule
- Requires quantitative approach
  - Profiling + Benchmarking
- Problem: Compiler writer doesn’t know the application beforehand

Summary

- Optimizations can improve speed, while maintaining correctness
- Various early optimization steps
- Global optimizations = dataflow analysis
- Reachability and Liveness analysis provides dataflow analysis
- Static Single-Assignment Form (SSA)